A Group Membership Algorithm with a Practical Specification *

Massimo Franceschetti Jehoshua Bruck

California Institute of Technology
Mail Code 136-93
Pasadena, CA 91125
Email: {massimo, bruck}@paradise.caltech.edu

(Revised Version, to Appear in IEEE Trans. Par. & Distr. Sys.)

June 28, 2001

Abstract

This paper presents a solvable specification and gives an algorithm for the Group Membership Problem in asynchronous systems with crash failures. Our specification requires processes to maintain a *consistent history* in their sequence of views. This allows processes to order failures and recoveries in time and simplifies the programming of high level applications. Previous work proved that the Group Membership Problem cannot be solved in asynchronous systems with crash failures. We circumvent this impossibility result building a weaker, yet non-trivial specification. We show that our solution is an improvement upon previous attempts to solve this problem using a weaker specification. We also relate our solution to other methods, and give a classification of progress properties that can be achieved under different models.

Keywords: distributed agreement algorithms, group membership, asynchronous systems.

*This work was supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, by an IBM Partnership Award by DARPA through an agreement with NASA/OSAT and by the Caltech Lee Center for Advanced Networking.

1 Introduction

Distributed systems consist of groups of processes that co-operate in order to complete specific tasks. A *Group Membership Protocol* is of particular use in such systems, providing processes in a group with a consistent view of the membership of that group. In this way, when a membership change occurs, processes can agree on which of them must complete a pending task, or start a new task. The problem of reaching a consistent membership view is very similar to the one of achieving common knowledge in a distributed system, commonly referred to as the *Consensus Problem*[39]. The Consensus Problem has been proven insolvable in asynchronous systems with crash failures [28].

Group Membership differs from Consensus in that the value to be agreed upon, namely the current membership view, may change due to asynchronous failures. Moreover, while Consensus requires *all* non-faulty processes to reach the same decision, Group Membership usually allows the removal of non-faulty processes from the group when they are erroneously suspected to have crashed, thus requiring agreement only on a subset of the processes in the system. Despite these differences, Chandra et al. [17] recently adapted the impossibility result for the Consensus Problem to the Group Membership Problem. At the same time, they conjectured that techniques used to circumvent the impossibility of Consensus can be applied to solve the Group Membership Problem. Such techniques include using randomization [8], probability assumptions on the behavior of the system [13], and using failure detectors that are defined in terms of *global* accuracy and completeness system properties [15].

It must be understood, however, that the impossibility result in this context really means "not always possible", as opposed to "never possible". As a matter of fact, the Chandra et al. result states that any algorithm that tries to solve the Group Membership Problem cannot always make progress; there are cases (although very unlikely) in which the algorithm blocks forever.

The above idea is the basis of Neiger's [48] approach, he suggests redesigning the problem by using a specification that is weak enough to be solvable — allowing the algorithm to block in some cases — but strong enough to prevent trivial implementations. His specification uses a weak progress requirement, allowing executions in which even a single process crash and its attempted removal from the membership may forever block all processes.

Our approach follows Neiger's intuition. We propose a specification that requires processes to install a new membership whenever they share a new view of the system connectivity. This requirement is weak, because if no set of processes agrees on the connectivity, no progress is made. The requirement is, however, stronger than the one proposed by Neiger, in that it implicitly states that removal and rejoining of any process must be allowed.

We summarize our contributions as follows: we identify the main assumptions required for proving the impossibility of Group Membership in asynchronous systems and relax one of them, namely the progress requirement, to break this result. Thus, we propose a weak, but not trivial, specification and a corresponding algorithm that solves this specification. Both specification and algorithm have the advantage of being simple. The specification we propose requires agreement on a sequence of views, that we call the consistent history requirement. Our consistent history definition requires that processes agree on the order in which the membership changes. This turns out to be a useful feature for many applications

(see for example the discussion in [22]). The specification also allows partitions to occur, and requires to maintain a consistent history within such partitions. Possible inconsistencies arising from partitioning may be solved in different ways: by using a primary partition mechanism based on a majority or quorum of processes [30] [38], by relying on network topologies resilient to partitions [37], or by implementing extended virtual synchrony [9] [46], using appropriate algorithms to merge the states when partitions are rejoined. Our algorithm, based on a standard three phase commit protocol, is fully distributed. It does not extend the asynchronous model of concurrent computation to include global failure detectors, and it can tolerate any number of removals and rejoining of processes. Progress of the algorithm can be easily guaranteed in practice in real world systems.

The rest of the paper is organized as follows. Section 2 discusses related work; Section 3 describes our model; Section 4 formally defines the specification; Section 5 relates our solution to other methods; Section 6 describes the protocol; Section 7 proves its correctness, and Section 8 draws conclusions and discusses some future work.

2 Historical Background

The Group Membership Problem first originated in a paper by Birman and Joseph [10]. Since then, a large amount of work has been done, in the context of synchronous [18], asynchronous under both a primary partition [4] [17] [29] [33] [41] [45] [48] [50] and a partitionable model [1] [2] [7] [24] [31] [34], and timed-asynchronous [19] [20] [21] [22] [26] [27] systems. A recent survey of these works appears in [54].

In general, specifications developed under asynchronous models are usually complex and many are difficult to understand. It appears that different protocols provide significantly different guarantees about their services, and are based on different assumptions of the system's behavior. Some specifications deal with more than just membership and views, and also consider message services with different ordering and reliability properties, as in [6] [7] [10] [25][27] [34]. Such specifications are known as Group Communication Services. This paper does not consider such extensions, but focuses on Group Membership alone.

As indication of how challenging specifying and implementing asynchronous membership algorithms is, widely cited research articles that attempt to give formal specifications for the primary partition and partitionable asynchronous membership problem [50] and [24], as well as their updated versions [25] [49] [51], contain some flaws in their formalisms. Anceaum et al. [3] showed that algorithms in [49] [50] and [51] allow undesirable executions and the specifications in [24] and [25] can be satisfied by trivial protocols.

Most of the difficulties in building a specification for the Group Membership Problem arise from the impossibility results in [17] and [28]. These results built a gap between group membership protocols that work in real systems — some of which have been around for many years, anticipating more theoretical results— and the formal specifications that they satisfy. Examples of such real systems that run group membership protocols are Amoeba [33] [52], Isis [11], Transis [23], Totem [47], Horus [53], Relacs [5], and more recently Phoenix [40] and RAIN [12]. Researchers developing these systems are, of course, aware of the original impossibility result [28] and of its potential application to their membership protocols [17], but they also believe that their systems can work under assumptions that can easily be

verified in practice. This motivates the study of formal specifications that are solvable in completely asynchronous settings. Yet, despite the wide interest it has attracted and the number of publications on this subject, it appears that currently there is no specification that is at the same time simple, practical, and does not rely on any extension to the asynchronous model of computation (e.g. the existence of global failure detectors).

Our work tries to fill this gap.

3 The Model

We consider an asynchronous distributed system, where processes communicate by exchanging messages. Processes are identified by unique *id*'s. The asynchronous model of execution of concurrent processes follows the one described in [35]. The communication model follows the one described in [32]. Every pair of processes is connected by a communication channel. That is, every process can send messages to and can receive messages from any other. We assume processes are able to probe a communication channel for incoming messages, using a boolean primitive as defined in [42]. Communication channels are considered to be reliable, FIFO, and to have an infinite buffer capacity. Message transmission and node processing times are finite but unpredictable, that is no upper or lower bounds are assumed on the execution speeds of the processes or on the delays experienced by messages in transit.

The failure model allows processes to crash, silently halting their execution. Because of the unpredictable delays experienced by the system, it is impossible to use time-outs to accurately detect a process crash. A process that has been infinitely slow for some time and has been unresponsive to other processes may become responsive again at any time. Therefore, processes can only *suspect* other processes to have crashed, using local failure detectors. Local failure detectors are assumed to be *inaccurate* and *incomplete*. That is, local failure detectors may erroneously suspect that other, operational processes have crashed, or that crashed processes are operational. Since local failure detectors run independently on each process, one local failure detector may perceive a failure, but other detectors may perceive it at a different time, or not at all.

We assume that a process communicates with its local failure detector through a special receive-only channel, on which the local failure detector may place a new list of *id*'s of processes not suspected to have crashed. We call this list the *local connectivity view* of the process. Each process considers the last local connectivity view received from its local failure detector as the current one.

We can summarize our model as follows:

- Sequential processes exchange messages on FIFO reliable channels with unbounded buffering capability and unpredictable delay, following the CSP specification in [32].
- Probes for incoming messages described in [42] extend the model in [32].
- The failure model includes process crashes. Local failure detectors inform processes of suspected changes in connectivity through special unidirectional communication channels.

4 Group Membership Specification

Each process p maintains two fundamental data structures: a set v_p, containing the current local connectivity view, and an infinite sequence $S_p = [V_{p1}, V_{p2}, \ldots V_{pk} \ldots]$ of global views. Initially all but the first view in S_p are empty, the first view being the initial state of the system. The problem is to extend the sequence of non-empty views, based on changes in the local views, and to do so consistently at different processes. Local connectivity views may change independently and arbitrarily at different processes, according to messages from local failure detectors. Sequences S_p must be extended at different processes maintaining a consistent global history in the sequence.

We define consistent history as follows:

Definition 1 Consistent History. A set of processes has a consistent history of views, if sequences S_p are the same at all processes, unless views V_{pj} and V_{qj} are disjoint. Namely:

$$ConsistentHistory \equiv \forall p, q, j \ (V_{pj} = V_{qj}) \ \vee \ (V_{pj} \cap V_{qj} = \emptyset)$$

where p and q are process *id*'s and V_{pj} and V_{qj} are the j-th elements in p's and q's global sequences of views, respectively.

We define a quiescent state as follows:

Definition 2 Quiescent State. A process p is in a quiescent state if it does not change its sequence of global views anymore. Namely:

$$QuiescentState_p \equiv \Box(S_p = \widehat{S_p})$$

where $\widehat{S_p}$ is a constant sequence of views, and $\Box$ is the *always* or *henceforth* operator [35], the notation $\Box A$ meaning that A holds at all time points after the reference point.

We make some remarks regarding the definitions above.

A quiescent state is stable, in the sense that once a process reaches a quiescent state, it stays in that state forever.

Consistent history requires all processes to have the same sequence of views, except for processes that were part of disjoint memberships (i.e. memberships that excluded each other); such processes are allowed to maintain the disjoint parts of their histories when they are connected again. Moreover, consistent history does not require processes that were excluded from all memberships and then readmitted to some membership, to have in their sequence views in which they did not participate.

We now define a specification, consisting of four properties, for a group membership algorithm. We assume the system to be initialized to a start state where the sequences S_p are the same at all processes, and the last non-empty views in their sequences are the ones reported by all failure detectors.

Property 1 Agreement. At any point in time all processes have a consistent history.

$$True \Rightarrow \Box(ConsistentHistory)$$

Property 2 Termination. If there are no more changes in the local views of the processes, they eventually reach their quiescent states

$$(\forall p \ \Box(v_p = \widehat{v_p})) \Rightarrow (\forall p \ \Diamond(QuiescentState_p))$$

where each $\widehat{v}_p$ is a constant set, and $\diamond$ is the *sometime* or *eventually* operator [35], the notation $\diamond A$ meaning that there is a time point after the reference point at which A holds.

Property 3 Validity. If all processes in a view v^* perceive view v^* as their local view, and they have reached their quiescent states, then the last non-empty elements of their sequences of global views are all at position j, and must be equal to v^*.

$$(\forall p \in v^* \Box(QuiescentState_p \wedge v_p = v^*)) \Rightarrow (\exists j \mid (\forall p \in v^*(j = \max_{V_{pk} \neq \emptyset} k \wedge V_{pj} = v^*)))$$

Property 4 Safety. Once a view is "committed" in the sequence of global views, it cannot be changed.

$$\forall p, j \ ((V_{pj} = v^* \neq \emptyset) \Rightarrow \Box(V_{pj} = v^*))$$

The first property expresses agreement. Consistent history must be an invariant for any program that satisfies the specification.

The second property expresses termination. When the inputs of all processes are stable, the processes are eventually going to stop changing their output sequences.

The third property rules out trivial solutions where protocols never decide on any new view or always decide on the same view. It ensures that a protocol that satisfies the specification does something useful, by stating that when all processes in a set agree on such set, they must commit this common view at the same position j in their sequences of global views. Note that this requirement is weak, because a new membership is created only if the local views of the different processes in the membership reach agreement.

The fourth property also rules out trivial solutions, requiring processes not to change old views in their sequences.

5 Circumventing the Impossibility Result

In this section, we relate our specification to other ways to solve the group membership problem, summarized in Table 1.

- In an asynchronous model augmented by global failure detectors, processes have access to modules that (by definition) eventually reflect the state of the system. Therefore, progress can be guaranteed unconditionally.

- In a timed asynchronous model, processes must react to an input, producing the corresponding output or changing state, within a known timebound. Under this model, progress can be guaranteed if no failures and recoveries occur for a known time needed to communicate in a timely manner.

- In a completely asynchronous model, progress cannot always be guaranteed and failure detectors in practice eventually reflect the system state, but they must be considered arbitrary. Correct processes react in practice within finite time, but this time cannot be quantified. Therefore, in order to guarantee a solution, we need a weaker specification of the problem.

Global Failure Detectors	*Timed Asynchronous*	*Completely Asynchronous*
Unconditional Progress	Conditional Timeliness	Weak Progress

Table 1: Classification of Progress Properties under Different Models

Our approach falls into the last category that originated with Neiger's work [48]. Our specification, however, differs from Neiger's in several ways.

- Processes in Neiger's model do not need to wait for convergence in their local views to change their membership. If one process suspects that another failed, it may attempt to remove the suspect process. Neiger's specification says that if one process attempts to remove another, they will eventually not be in the same membership. Our specification requires all processes in a set to agree on that set before changing their membership.

- Neiger's specification allows a solution in which the attempted removal of a single process blocks all processes. Our specification does not allow such a single point of failure, because it states that if all processes in a set agree on such set, they must eventually commit such set.

- Finally, Neiger's specification does not consider processes rejoining the group. It states that the membership changes only by processes leaving the group. Our specification allows removal and rejoining of any number of processes.

We now relate our solution to the other methods quoted in table 1.

In a *timed* asynchronous model, processes must react to an input, producing the corresponding output or changing state, within a *known* timebound. Progress under this model is achieved if there are no failures or recoveries in the system for an a priori known duration. This implies a certain amount of synchronism, which is absent from our totally asynchronous model.

Global Failure detectors strengthen the asynchronous time-free model, based on the observation that the system eventually stabilizes: this essentially implies that, after a "sufficiently long" time, failure detectors are accurate and complete, thus reflecting the actual system state. In our model stability is also required for progress; but, at variance of the above case, it is not necessarily related to the state of the system. In other words, eventual progress is required when there is agreement among a subset of the local failure detectors, even if failures and recoveries continue to occur in the system.

If our local failure detectors do not necessarily reflect the system state, because they are inaccurate and incomplete, how is it possible to require progress in a group that agrees on a wrong connectivity? For example, if processes p_1, p_2 and p_3 agree on the view $\{p_1, p_2, p_3\}$, but process p_3 has stopped and cannot communicate with p_1 and p_2, how can p_3 commit the view $\{p_1, p_2, p_3\}$? There are two answers to this question. First, p_3 may remain stopped for a while (or just be infinitely slow) and may become responsive again at any time. In this case, if the three processes do not change their local view and since channels are reliable, all three will be able to commit the view $\{p_1, p_2, p_3\}$ when communication with p_3 is restored. This

is enough to guarantee *eventual* progress. Second, if p_3 has crashed permanently (consider for example turning off a computer), it does not have a local view and does not agree on the view $\{p_1, p_2, p_3\}$. Therefore the specification does not require the two remaining processes to commit such view.

In practice, local failure detectors are typically implemented using timeouts and "heartbeats" and are at least complete, if not accurate, since all crashed processes eventually time out. In the example above, the permanently crashed process p_3 would be timed out by the remaining two processes and those processes would be able to commit the view $\{p_1, p_2\}$. However, in our specification we ignore global properties of failure detectors and consider their reports to be as random inputs.

6 Group Membership Algorithm

We provide an algorithm that solves the Group Membership Specification given in Section 4.

The algorithm is based on the three asynchronous phases depicted in Figure 1.

- A preparation phase, in which a process proposes a new view that matches the view of the other processes.
- A ready phase, in which all processes that agree on the new view acknowledge the reservation of a position in their sequence of global views to commit such view.
- A commit phase, in which the new view is finally installed, and the sequences S_p of global views are extended consistently at different processes.

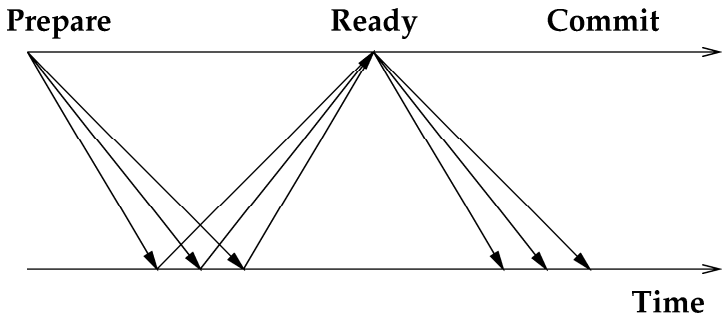

Figure 1: Phases of the algorithm

6.1 Solution Sketch

The main idea for the algorithm is as follows: a process p that is informed by its local failure detector of a change in its local connectivity view and that has the smallest *id* among processes in its new local connectivity view, sends a message to all processes in its view, proposing to update the current membership with the new view. Each process records this proposal until its local view is the same as the proposed view, at which point it responds

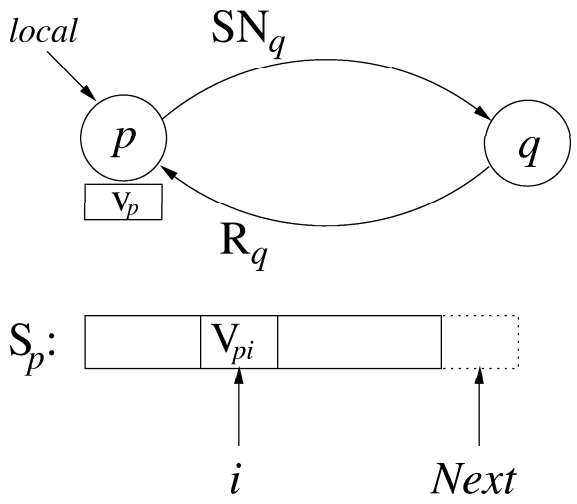

Figure 2: Channels and data structures

by sending back an *Accept* or *Retry* message to the process that proposed the membership update. The *Accept* message is sent if the process agrees on the proposed group index, namely on the position in its sequence S_p where to place the new view. Upon sending the *Accept* message, the process reserves the corresponding position in its sequence S_p, so that no other proposal is accepted for that group index. Upon receiving a *Retry* message, the proposing process restarts the first phase of the algorithm, sending a new *Propose* message to all processes in its view, this time with a new group index. When the proposing process has collected *Accept* messages from all processes in its view, it starts the commit phase, by sending *Commit* messages, ordering other processes in its view to commit the membership update. Upon receiving a commit message, processes extend their sequences S_p accordingly.

6.2 Channels and Data Structures

Our communication model is depicted in Figure 2. A process p is connected to a process q through a send channel SN_q and a receive channel R_q. Process p also has a receive channel named *local*, coming from its local failure detector.

The local view of process p is stored in the variable v_p. A global view that has been committed by process p at position i in its sequence S_p of global views is represented by V_{pi}. The index *Next* of process p always points to the first position in the sequence S_p where a new view can be committed. The remaining local variables of process p are summarized in Table 2.

6.3 CSP Notation

We specify our algorithm using Hoare's CSP [32]. A full description of our notation and its semantics can be found in [43] and [44]. What follows is a short summary of the notation we use.

Variables of process p	
p	self index
v_p	local view
V_{pi}	global view at position i
$Next$	next available position to commit a view
v^*	view of most recent proposal sent
$PropOut$	index of most recent proposal sent
$Prop[q]$	whether q proposed a view
$PropIn[q]$	index proposed by q
$v[q]$	view proposed by q
$ack[q]$	whether q has accepted p's view
$old, new, view, i$	temporary variables

Table 2: Variables of process p

- Statements:
  - Assignment: $a := b$.
  - Send: $X!e$ means send the value of e over channel X.
  - Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
  - Probe: The boolean expression $\overline{X}$ is *true* iff a Receive statement over channel X can complete without suspending.
- Control Structures:
  - Selection: $[G_1 \rightarrow S_1[] \ldots []G_n \rightarrow S_n]$, where G_i's are boolean expressions (guards) and S_i's are program parts. The execution of this command corresponds to waiting until one of the guards is *true*, and then executing one of the statements with a *true* guard.
  - Repetition: The notation $*[S]$ means repeat S forever.
  - Sequential execution: S; T.
  - Parallel execution: $S \parallel T$.
  - Parameterization: The notation $\langle []_{q:q \in U} S \rangle$ means: $S_{q1}\ []S_{q2}\ []S_{q3} \ldots$, where S_{qi} is the program part S with q replaced by the ith member of the set U. The notation $q ::$ is a shorthand for $q : q \in \{\text{set of all processes}\}$.

6.4 Code Description

The code is shown in Figure 3. The first guarded command in Figure 3 shows how a process p, when informed of a change in its local connectivity view, checks if it has the minimum id among the processes in v_p. If p has the minimum id, it proposes to extend the sequence of global views at position $PropOut$ with the view v_p, by broadcasting v_p and $PropOut$ to all processes in v_p, and it initializes its ack array to zero.

```
*[[ local ⟶ local?(v_p);
            [ p = min(v_p) ⟶ PropOut := PropOut + 1;
                             v* := v_p;
                             ⟨||_{q:q∈v*} SN_q!Propose(v*, PropOut)⟩;
                             ⟨||_{q::} ack[q] := 0⟩
            [] else ⟶ skip
            ]

 [] ⟨[]_{q::
      R_q ⟶ R_q?m;
            [ m.type = Propose⟶ (v[q], PropIn[q]) := m;
                                Prop[q] := true

            [] m.type = Retry⟶ (old, new) := m;
                               [PropOut = old ∧ p = min(v_p) ⟶ PropOut := new;
                                                               ⟨||_{q:q∈v_p} SN_q!Propose(v_p, PropOut)⟩;
                                                               ⟨||_{q::} ack[q] := 0⟩
                               [] else ⟶ skip
                               ]

            [] m.type = Acc⟶ i := m;
                             [PropOut = i ⟶ ack[q] := 1
                             [] else ⟶ skip
                             ]
                             [∀_{q:q∈v*} ack[q] = 1 ⟶ ⟨||_{q:q∈v*} SN_q!Commit(v*, PropOut)⟩;
                                                      ⟨||_{q::} ack[q] := 0⟩
                             [] else ⟶ skip
                             ]

            [] m.type = Commit⟶ (view, i) := m;
                                V_pi := view
            ]

   [] Prop[q] ∧ (v[q] = v_p) ⟶ Prop[q] := false;
                               [ PropIn[q] < Next ⟶ SN_q!Retry(PropIn[q], Next)
                               [] PropIn[q] ≥ Next ⟶ Next = PropIn[q] + 1;
                                                     SN_q!Acc(PropIn[q])
                               ]

⟩]]
```

Figure 3: The algorithm

The second guarded command in Figure 3 checks for incoming messages from other processes. These may be proposals for a new membership (*Propose*), invitations to retry proposing a membership with a new group index (*Retry*), acceptances of a proposed membership (*Acc*), or orders to commit a new membership (*Commit*).

Upon receiving a proposal message from process q, process p stores the view proposed by q at position q of the array v, and stores the proposed group index at position q of the array $PropIn$, then sets position q of the array *prop* to true, to record the receipt of the proposal from q.

If process p later agrees on the proposed view, it sends a response to process q (see last guarded command in Figure 3). The response is either an acceptance of the view $v[q]$ at position $PropIn[q]$, if the next available slot of process p to commit a view is at a position that is less or equal than the proposed index $PropIn[q]$; or it is an invitation to retry, with a different index, if the next available slot of process p is at a position that is greater than the proposed index. An invitation to retry consists of sending back to q the proposed index and the current next available index, to be used for the retry. An acceptance consists of acknowledging the proposed view at position $PropIn[q]$. Following an acceptance, process p also increments its index $Next$ to the value $PropIn[q] + 1$, in order not to accept any other proposal for that position.

We now examine the guarded commands of the remaining message types.

A process p that receives an invitation to retry its proposal, receives two indices, *old* and *new*. The former is the one p has sent in the proposal, the latter is the one p will use in its next proposal. If index *old* is equal to the index of the most recent proposal sent by process p, and process p still has the minimum *id* among the processes in its local view, it proposes to extend the sequence of global views at the new position *new*, with the view v_p, and it re-initializes the *ack* array to zero.

A process p that receives an acceptance regarding its proposed view, receives an index i. If i is equal to the index of the most recent proposal sent, process p sets the element at position q in the array *ack* to 1 to record the acceptance. Then it inspects the *ack* array to check if all entries are 1. If so, p starts the commit phase by broadcasting its previously proposed view v^* and the corresponding proposed index *PropOut* to all processes in v^* and it re-initializes the *ack* array to zero.

A process p that receives an order to commit a view at position i from process q, simply sets the view at position i of its sequence of global views to the received view.

6.5 Local Failure Detectors Implementation

Our algorithm relies on an implementation of the local failure detectors that satisfies the following property:

- Given any pair of processes p and q, every time p suspects q failed, it cannot un-suspect q, until q has also suspected p failed.

Note that such an implementation does not introduce any global property in the failure detection mechanism. In fact, it does not relate suspected failures to actual failures in the system in any way.

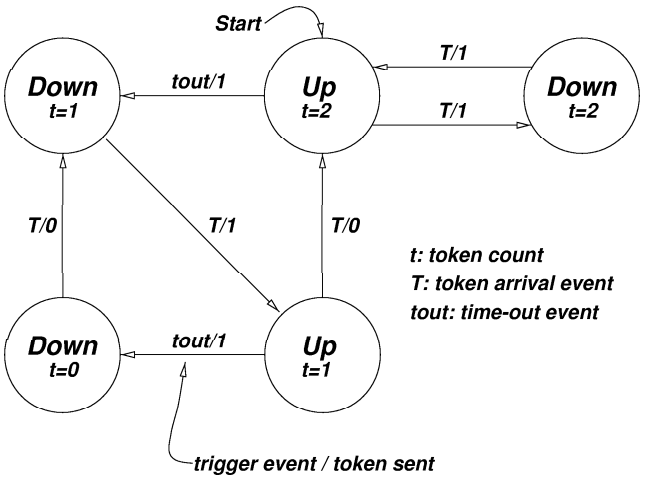

Figure 4: State machine for process p's local view of process q.

It is easy to implement local failure detectors that fulfill this requirement. In particular, an implementation of a very simple protocol for this kind of local failure detectors is reported in [12] and [36], and its state machine description is depicted in Figure 4. Instead of replicating this state machine into the CSP description of our algorithm, we assume to have incorporated it within the local failure detector implementation.

The state machine depicted in Figure 4 shows process p's local view of process q. It shows the reaction to t_{out} events and T (token-receipt) events by process p that is at one end of the communication channel to process q. The protocol consists of two parts:

- First, we have the sending and receiving of tokens, using reliable messaging. Tokens are sent whenever process p sees a state transition over the channel to process q, i.e., it suspects or un-suspects process q.
- Second, we have an (unreliable) hint from the underlying system, such as a time-out, that indicates that communication to process q has (perhaps) been lost.

In Figure 4, each state is characterized by whether process p considers process q to be Up or $Down$, and by how many tokens are held by process p. The state transitions are labeled by the action triggering the transition and by the action taken upon transition. A trigger event is either a time-out t_{out}, or receipt of a token T. The action taken is always whether the token is sent (1) or not (0). Note that a token T is sent whenever a transition for an Up state to a $Down$ state or from a $Down$ state to an Up state is made.

6.6 Example

In order to clarify the behavior of the overall algorithm, we show the following example.

Let us consider the four processes system depicted in Figure 5 as a complete graph with four nodes. Initially (step (a) of Figure 5), all four nodes have the same view sequence $S = \{V_1, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \dots\}$ with $V_1 = \{1, 2, 3, 4\}$. Suppose node 1 is disconnected from the

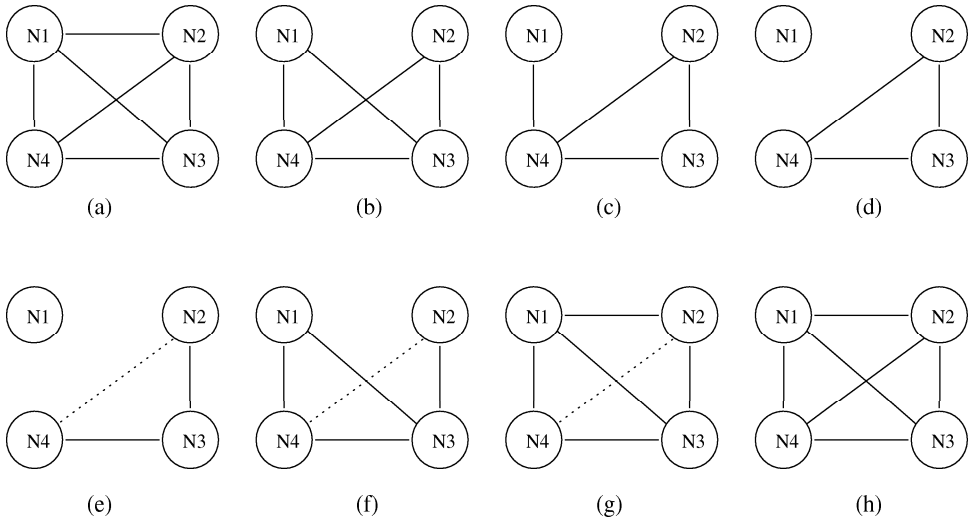

Figure 5: Example

network and suppose that local failure detectors of different nodes discover the failure at different times, reporting the connectivity views represented in steps (b) through (d).

At Step (b), node 1 suspects 2 has failed and node 2 suspects 1 has failed. Accordingly, at Step (b), both nodes 1 and 2 have the minimum id among their local views. Therefore, node 2 proposes the new membership $v = \{2, 3, 4\}$, tagged with group index $i = 2$, while node 1 proposes $v = \{1, 3, 4\}$, also tagged with group index $i = 2$. At Step (b) nodes 3 and 4 do not send any response, because their local views are still $v = \{1, 2, 3, 4\}$.

At Step (c), node 1 suspects 2 and 3 have failed and nodes 2 and 3 suspect 1 has failed. Accordingly, at Step (c), node 1 proposes $v = \{1, 4\}$, tagged with group index $i = 3$, while node 3 accepts the view $v = \{2, 3, 4\}$ proposed by node 2 at Step (b). Finally, at Step (d), node 4 also accepts the view $v = \{2, 3, 4\}$ proposed by node 2 at Step (b), while node 1 proposes and commits the singleton view $v = \{1\}$ at position $i = 4$.

By Step (d), node 2 has collected *Accept* messages from all nodes it sent its proposal to, therefore it is able to start the commit phase. Suppose that the *Commit* message sent to node 4 is delayed (Step (e)), so that nodes 2 and 4 suspect each other to have failed. Therefore, by Step (e) the sequences of views at the different nodes are:

- Node 1. $S = \{\{1, 2, 3, 4\}, \{\emptyset\}, \{\emptyset\}, \{1\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \dots\}$
- Nodes 2,3. $S = \{\{1, 2, 3, 4\}, \{2, 3, 4\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \dots\}$
- Node 4. $S = \{\{1, 2, 3, 4\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \dots\}$

Now suppose that node 1 is reconnected to the system (steps (f) and (g)). Accordingly, node 1 proposes $v = \{1, 3, 4\}$ with group index 5 at step (f), while node 2 proposes $v = \{2, 3\}$ with group index 3. Node 3 does not respond to any these proposals, because its local view is $v = \{1, 2, 3, 4\}$. Node 4, sends an *Accept* message to node 1, regarding the new proposed view $v = \{1, 3, 4\}$ at position 5, but the commit phase does not start for this view. Node 1

then proposes $v = \{1, 2, 3, 4\}$ with group index 6 at step (g). Node 3 accepts this proposal at Step (g); nodes 2 and 4 accept it at Step (h). Node 4 also receives the delayed commit message from node 2 at Step (h). By Step (h), node 1 has collected *Accept* messages from all the nodes it sent its proposal to, therefore it starts the commit phase. Supposing node 1 completes the commit phase at Step (h), the final sequences of views at the different nodes become:

- Node 1. $S = \{\{1, 2, 3, 4\}, \{\emptyset\}, \{\emptyset\}, \{1\}, \{\emptyset\}, \{1, 2, 3, 4\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \dots\}$
- Nodes 2,3,4. $S = \{\{1, 2, 3, 4\}, \{2, 3, 4\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \{1, 2, 3, 4\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \dots\}$

7 Correctness

Correctness of the algorithm is ensured by proving that it satisfies the four properties of the specification given in Section 4.

7.1 Agreement

Theorem 1 *The algorithm described in Section 7 satisfies the agreement condition of the specification (Property 1, Section 4): at any point in time, all processes have a consistent history.*

Proof of Theorem 1. Either all processes remain in the start state, or some process p extends its sequence S_p of global views. In the start state the consistent history property holds. If some process p extends its sequence S_p by committing a new view v^* at a given position i, it must have received a *Commit* message from some process q, therefore q must have received *Accept* messages, regarding its proposal of v^* at position i, from all processes in v^*, including p. It follows from the last guarded command in Figure 3, that if process p has accepted the proposal of process q, it has also increased its $Next$ variable to the value of $i+1$, and will not accept any other proposal for that position. Therefore, process p either commits view v^* at position i, or ends up with position i of its sequence of global views empty. The consistent history property follows. □

7.2 Termination

Theorem 2 *The algorithm described in Section 7 satisfies the termination condition of the specification (Property 2, Section 4): all processes eventually reach a quiescent state, if there are no more changes in their local views.*

Proof of Theorem 2. By contradiction, a non-quiescent state means that the sequence of global views is extended infinitely often at some process, therefore an infinite number of *Commit* messages must be sent. Since the number of processes is finite, there must be at least one process that sends infinitely many *Commit* messages. Call this process p. By the code in Figure 3 we see that each time p sends a commit message, it re-initializes its *ack* array to 0. It follows that, in order to send infinitely many *Commit* messages, process p must re-fill the array with 1's, infinitely often. Since each process sends at most one *Acc*

message for each proposal, process p must send infinitely many proposals. Proposals are sent either when there is a change in the local view, or because of an invitation to retry has been received (see Figure 3). The first case is ruled out, because it implies that process p changes its local view infinitely often, and by hypothesis there are no more changes in the local views of the processes. The second case is also ruled out, because it implies that process p sends infinitely many proposals of the same view, each time increasing the proposed index. Since the number of processes is finite, and there are no more changes in their local views, process p must eventually propose an index large enough, that does not generate any *Retry* messages, leading to the desired contradiction. □

7.3 Validity

Theorem 3 *The algorithm described in Section 7 satisfies the validity condition of the specification (Property 3, Section 4): if all processes in a view v^* perceive view v^* permanently as their local view, and if they have reached their quiescent states, then they must have installed view v^* at position j, as the last non-empty element of their sequences of global views.*

Proof of Theorem 3. Let all processes in v^* perceive view v^* permanently as their local view. We first show that when the processes change their local views to v^*, process $p = min(v^*)$ proposes view v^*. We distinguish two cases:

- **Case 1.** All processes in v^* always perceive p as not failed. In this case, since $p = min(v^*)$, the only process that can ever send a proposal is process p. Moreover, by the assumption on the behavior of the local failure detectors (see Section 6.5), p can never suspect any $q \in v^*$ and then perceive v^* again. It follows that either p proposes v^* by excluding some process that is not in v^*, or p never proposes v^*, in which case v^* must be the initial view of all processes in the system.

- **Case 2.** There is at least a process $q \in v^*$ that perceives a transition from considering p failed to considering p not failed. In this case, by the assumption on the local failure detectors (see Section 6.5), p must suspect q before q un-suspects p. It follows that, to perceive v^*, p must at some time perceive a transition to un-suspect q, and propose a view letting q join the group. The last process q for which process p perceives such a transition makes p send a proposal for view v^*.

When process p sends a proposal for view v^*, and all processes in v^* agree on such view, the proposal is either accepted or an invitation to retry is sent by some process back to process p. Since for every *Retry* message received, process p sends a new proposal for v^*, with a larger proposed index, and since the local views of all processes are stable, process p must eventually propose an index that is large enough to be accepted by all processes in v^*, and therefore must eventually send *Commit* messages to all processes in v^*. It follows, that by the time they reach a quiescent state, all processes in v^* have installed view v^* at the same position in their sequences of global views. □

7.4 Safety

Theorem 4 *The algorithm described in Section 7 satisfies the safety condition of the specification (Property 4, Section 4): once a view is "committed" in the sequence of global views, it cannot be changed.*

Proof of Theorem 4. Every time a process sends an *Acc* message, regarding a view at a given position i, it increases the value of its $Next$ variable to $i + 1$, therefore, it will not accept any other proposal for position i, making it possible to commit at most a single view at position i. □

8 Conclusion

We have presented a specification for the Group Membership Problem in completely asynchronous systems, and a corresponding algorithm that solves it.

Our specification requires processes to maintain a *consistent history* in their sequence of views. This allows processes to order failures and recoveries in time and simplifies the programming of many high level applications (see for example the discussion in [22]).

We have assumed our local failure detectors to be inaccurate and incomplete. With this approach, the specification states explicitly that progress cannot always be guaranteed.

In practice, our requirement for progress is weaker than that stated in the specification of having a set of processes sharing the same connectivity view indefinitely. In fact, if the rate of perceived failures in the system is lower than the time it takes the protocol to make progress and commit a new membership, then it is possible for the algorithm to make progress every time there is a failure in the system. This depends on the actual rate of failures and on the capacity of the failure detectors to track such failures.

Reference [15] notes that failure detectors defined in terms of global system properties cannot be implemented. This result gives strength to the approach of relaxing the specification and of having a protocol in continuous search for convergence. In *real world* systems, where process crashes actually lead a connected cluster of processes to share the same connectivity view of the network, convergence on a new membership can be easily reached in practice.

In the presence of partitions, our protocol does not attempt to solve inconsistencies in the history of views of processes that commit disjoint memberships. Therefore, two processes that partition and evolve separately as part of different components maintain separate histories in their view sequences. This information may be used later by other algorithms to place a total order on the memberships of the whole network.

We believe that the weak liveness approach, introduced by Neiger [48], and expanded here, can be generalized to other agreement problems in asynchronous systems.

9 Acknowledgments

The authors would like to thank prof. A. J. Martin and his student Robert Southworth, who helped formalize the problem and gave suggestions on how to specify the code using CSP;

Michael Gibson, who read a previous version of the manuscript and made useful comments to improve the presentation; and all the anonymous referees for their constructive criticism.

Extra thanks are due to the anonymous referee who pointed out an error in a previous version of the paper that allowed us to simplify the algorithm; and to Matthew Cook, for reviewing the final version of the manuscript.

References

[1] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal and P. Ciarfella. "The Totem Single-Ring Ordering and Membership Protocol". *ACM Transactions on Computer Systems 13-4 pp. 311-342, November 1995.*

[2] Y. Amir, D. Dolev, S. Kramer and D. Malki. "Membership Algorithms for Multicast Communication Groups". *Proceedings of the Sixth International Workshop of Distributed Algorithms (Lecture Notes in Computer Science 647), pp. 292-312, November 1992.*

[3] E. Anceaume, B. Charron-Bost, P. Minet and S. Toueg. "On the Formal Specification of Group Membership Services". *Technical Report 95-1534, Computer Science Department, Cornell University, August 1995.*

[4] T. Anker, G.V. Chockler, D. Dolev and I. Keidar. "Scalable Group Membership Services for Novel Applications". *Proceedings of the Workshop on Networks in Distributed Computing (DIMACS 45), pp. 23-42, AMS, 1998.*

[5] O. Babaöglu, R. Davoli, L. Giachini and M. Baker. "Relacs: A communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems". *Proceedings of Hawaii International Conference on Computer System Science. Vol. 2, pp. 612-621, 1995.*

[6] O. Babaöglu, R. Davoli and A. Montresor. "Failure Detectors, Group Membership and View Synchronous Communication in Partitionable Asynchronous Systems". *Technical Report UBLCS–95-18, Computer Science Department, University of Bologna, Italy, November 1995.*

[7] O. Babaöglu, R. Davoli and A. Montresor. "Group Communication in Partitionable Systems: Specifications and Algorithms". *Technical Report UBLCS–98-01, Computer Science Department, University of Bologna, Italy, October 1999.*

[8] M. Ben-Or. "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols". *Proceedings of the Second ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 27-30, August 1983.*

[9] K. Birman. "Building Secure and Reliable Network Applications". *Manning Publications Co. 1996.*

[10] K. Birman and T. Joseph. "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems, 5-1 pp. 47-76, February 1987.*

[11] K. Birman and R. van Renesse. "Reliable Distributed Computing with the ISIS Toolkit". *IEEE Computer Society Press, 1994.*

[12] V. Bohossian, C. Fan, P. LeMahieu, M. Riedel, L. Xu and J. Bruck. "Computing in the RAIN: A Reliable Array of Independent Nodes". *IEEE Transactions on Parallel and Distributed Systems, 12-2 pp.97-113, February 2001.*

[13] G. Brancha and S. Toueg. "Resilient Consensus Protocols". *Proceedings of the Second ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 12-26, August 1983.*

[14] R. Carr. "The Tandem Global Update Protocol". *Tandem Systems Review, June 1985.*

[15] T. D. Chandra and S. Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". *Journal of the ACM 43-2 pp. 225-267, March 1996.*

[16] T. D. Chandra, V. Hadzillacos, S. Toueg. "The Weakest Failure Detector for Solving Consensus". *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 147-158, August 1992.*

[17] T. D. Chandra, V. Hadzillacos, S. Toueg and B. Charron-Bost. "On the Impossibility of Group Membership". *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 322-330, May 1996.*

[18] F. Cristian. "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems". *Distributed Computing, 4 pp. 175-187, April 1991.*

[19] F. Cristian. "Probabilistic Clock Synchronization". *Distributed Computing, 3 pp. 146-158, 1989.*

[20] F. Cristian. "Synchronous and Asynchronous Group Communication". *Communications of the ACM, 39-4 pp. 88-97, 1996.*

[21] F. Cristian and C. Fetzer. "The Timed Asynchronous Distributed System Model". *IEEE Transactions on Parallel and Distributed Systems, 10-6 pp. 642-657, June 1999.*

[22] F. Cristian and F. Schmuck. "Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems". *Technical Report CSE95-428, Computer Science Department, University of California at San Diego, 1995.*

[23] D. Dolev and D. Malki. "The Transis Approach to High Availability Cluster Communication". *Communications of the ACM, 39-4 pp.64-70, 1996.*

[24] D. Dolev, D. Malki and R. Strong. "An Asynchronous Membership Protocol that Tolerates Partitions". *Technical Report CS94-6, Computer Science Department, The Hebrew University of Jerusalem, Israel, 1994.*

[25] D. Dolev, D. Malki and R. Strong. "A Framework for Partitionable Membership Services". *Technical Report CS95-4, Computer Science Department, The Hebrew University of Jerusalem, Israel, 1995.*

[26] C. Dwork, N. Lynch and L. Stockmeyer. "Consensus in the Presence of Partial Synchrony". *Journal of the ACM, 35-2 pp. 288-323, 1988.*

[27] A. Fekete, N. Lynch and A. Shvartsman. "Specifying and Using a Partitionable Group Communication Service". *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 53-62, August 1997.*

[28] M. J. Fischer, N. A. Lynch and M. S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". *Journal of the ACM 32-2 pp. 374-382, April 1985.*

[29] M. Franceschetti and J. Bruck. "On the Possibility of Group Membership". *Proceedings of the IEEE Workshop on Fault Tolerant Parallel and Distributed Systems. San Juan, Puerto Rico, April 1999. Published in D.E. Avresky Editor, Dependable Network Computing, Chapter 4, pp.77-92, Kluwer Academic.*

[30] S. Jajodia and D. Mutchler. "Dynamic Voting for Maintaining the Consistency of a Replicated Database". *ACM Transactions on Database Systems 15-2 pp. 230 - 280, June 1990.*

[31] F. Jahanian, S. Fakhouri and R. Rajkumar. "Processor Group Membership Protocols: Specification, Design and Implementation. *Proceedings of the Twelfth Symposium on Reliable Distributed Systems, October 1993.*

[32] C.A.R. Hoare. "Communicating Sequential Processes". *Communications of the ACM 21-8 pp. 666-677, 1978.*

[33] F. Kaashoek and A. Tanenbaum. "Group Communication in the Amoeba Distributed System". *Proceedings of the Eleventh International Conference on Distributed Computing Systems, pp. 222-230, May 1991.*

[34] I. Keidar, J. Sussman, K. Marzullo and D. Dolev. "A Client Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs". *Proceedings of the twentieth International Conference on Distributed Computing Systems. Taipei, Taiwan, April 2000.*

[35] F. Kröger. "Temporal Logic of Programs". *Springer Verlag, 1987.*

[36] P. LeMahieu and J. Bruck. "A Consistent History Link Connectivity Protocol". *Proceedings of the Seventeenth ACM Symposium on Principles of Distributed Computing, ACM Press, p. 309, July 1998 (extended abstract).* Full version paper appeared in the *Proceedings of the Thirteenth International Parallel Processing Symposium (IPPS 99), pp. 138-142, April 1999.*

[37] P. LeMahieu, V. Bohossian and J. Bruck. "Fault-Tolerant Switched Local Area Networks". *Proceedings of the Twelfth International Parallel Processing Symposium (IPPS 98), pp.747-751, April 1998.*

[38] E. Y. Lotem, I. Keidar and D. Dolev. "Dynamic Voting for Consistent Primary Components". *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 63-71, August 1997.*

[39] N. Lynch. "Distributed Algorithms". *Morgan Kaufman, 1996.*

[40] C.P. Malloth, P. Felher, A. Shiper and U. Wilhelm. "Phoenix: A Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks". *Proceedings of the Workshop on Parallel and Distributed Platforms in Industrial Products. San Antonio, Texas, October 1995.*

[41] C.P. Malloth and A. Shiper. "View Synchronous Communication in Large Scale Distributed Systems". *Proceedings of the Second Open Workshop of the ESPRIT Project. Grenoble, France, July 1995.*

[42] A.J. Martin. "The Probe: An Addition to Communication Primitives". *Information Processing Letters 20 pp.125-130, 1985.*

[43] A.J. Martin. "Compiling Communicating Processes into Delay-insensitive VLSI circuits". *Distributed Computing, 1-4, pp.226-234, 1986.*

[44] A.J. Martin. "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits". *C.A.R. Hoare Editor, Developments in Concurrency and Communication, Chap. 1, Addison Wesley, 1990.*

[45] P.M. Melliar-Smith, L. E. Moser and V. Agrawala. "Processor Membership in Asynchronous Distributed Systems". *IEEE Transactions on Parallel and Distributed Systems, 5-5 pp. 459-473, May 1994.*

[46] L. E. Moser, L.E. Amir, P.M. Melliar-Smith and D. A. Agarwal. "Extended Virtual Synchrony". *Proceedings of the Fourteenth IEEE International Conference on Distributed Computing Systems, pp. 55-65, June 1994.*

[47] L. E. Moser, P.M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos. "Totem: A Fault Tolerant Multicast Group Communication System". *Communications of the ACM, 39-4 pp. 54-63, 1996.*

[48] G. Neiger. "A New Look at Membership Services". *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 331 - 340, May 1996.*

[49] A. M. Ricciardi. "The Group Membership Problem in Asynchronous Systems". *Ph.D. Thesis, Department of Computer Science, Cornell University, 1993.*

[50] A. M. Ricciardi and K. Birman. "Using Process Groups to Implement Failure Detection in Asynchronous Environments". *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing, ACM Press, pp. 341-352, May 1991.*

[51] A. M. Ricciardi and K. Birman. "Process Membership in Asynchronous Environments". *Technical Report, Department of Computer Science, Cornell University, 1995.*

[52] A. Tanenbaum, R. van Renesse, H. Vanstaveren, G. J. Sharp, S. J. Mullender, J. Jansen and G. Vanrossum. "Experiences with the Amoeba Distributed Operating System". *Communications of the ACM 33-12 pp. 46-63, December 1990.*

[53] R. van Renesse, K. Birman and S. Maffeis. "Horus: A Flexible Group Communication System". *Communications of the ACM, 39-4 pp. 76-83, 1996.*

[54] R. Vitenberg, I. Keidar, G. Chockler and D. Dolev. "Group Communication Specifications: A Comprehensive Study". *Technical Report CS99-31, Institute of Computer Science, The Hebrew University of Jerusalem, September 1999.*

Massimo Franceschetti was born in Naples, Italy, and received his education in Italy, U.K., and U.S.A. He received the Laurea degree, magna cum laude, in computer engineering, from University of Naples, Italy, in June 1997. During his studies he spent one semester at the University of Edinburgh, U.K., as a visiting student, thanks to a fellowship by the Students Award Association of Scotland. He received the MSc degree in electrical engineering from the California Institute of Technology in June 1999. He is currently a Ph.D. student in electrical engineering at the California Institute of Technology, advised by Dr. Bruck. He is a recipient of the 1999 UPE/IEEE Computer Society award for academic excellence, and of the 2000 Caltech Walker von-Brimer Foundation award for outstanding research initiative. His research interests include mathematical models of wireless networks, applied probability, and fault tolerant distributed computing.

Jehoshua Bruck received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion, Israel Institute of Technology, in 1982 and 1985, respectively and the Ph.D. degree in electrical engineering from Stanford University in 1989. He is the Moore Professor of Computation and Neural Systems and Electrical Engineering at the California Institute of Technology. His research interests include parallel and distributed computing, fault-tolerant systems, error-correcting codes, computation theory and biological systems. Dr. Bruck has an extensive industrial experience, including, with IBM for ten years both at the at the IBM Almaden Research Center and the IBM Haifa Science center. Dr. Bruck is a co-founder and Chairman of Rainfinity, a spin-off company from Caltech that is focusing on providing software for high performance reliable Internet infrastructure. Dr. Bruck is the recipient of a 1997 IBM Partnership Award, a 1995 Sloan Research Fellowship, a 1994 National Science Foundation Young Investigator Award, six IBM Plateau Invention Achievement Awards, a 1992 IBM Outstanding Innovation Award for his work on "Harmonic Analysis of Neural Networks" and a 1994 IBM Outstanding Technical Achievement Award for his contributions to the design and implementation of the SP-1, the first IBM scalable parallel computer. He published more than 150 journal and conference papers in his areas of interests and he holds 22 patents. Dr. Bruck is a Fellow of the IEEE.